

**Efficient and Correct Execution of
Parallel Programs That Share Memory**

by

Dennis Shasha

Marc Snir†

Ultracomputer Note #96

Technical Report #206

March, 1986



Ultracomputer Research Laboratory

NYU COMPSCI TR-206 c.2
Shasha, Dennis
Efficient and correct
execution of parallel
programs that share memory

New York University
Courant Institute of Mathematical Sciences
Division of Computer Science
251 Mercer Street, New York, NY 10012



**Efficient and Correct Execution of
Parallel Programs That Share Memory**

by

Dennis Shasha

Marc Snir[†]

Ultracomputer Note #96

Technical Report #206

March, 1986

[†] Institute of Mathematics and Computer Science, Hebrew University of Jerusalem.

The work of the first author was partially supported by the National Science Foundation under grant number DCR8501611 and by the Office of Naval Research under grant number N00014-85-K-0046. The work of the second author was done while the author was in residence at the Courant Institute.

ABSTRACT

In this paper, we consider an optimization problem that arises in the execution of parallel programs on shared memory multiple-instruction stream multiple-data stream (MIMD) computers. A program on such a machine consists of many *program segments* each executed sequentially by a single processor. The processors have access to shared memory, and can execute standard memory access operations on this shared memory. This memory is distributed among many separate memory modules. A network connects processors to memory modules. Delays on this network are stochastic. Thus, operations issued by a processor to distinct memory modules may not be executed as memory requests on those modules in the order they were issued.

For performance reasons, we want to allow one operation to begin before a previous one in the same instruction. Our analysis gives a method for determining which operations in a stream may be issued concurrently without changing the semantics of the execution. We also consider code where blocks of operations have to be executed atomically. This introduces the necessity of locks. We use a conflict graph similar to that used to schedule transactions in distributed databases. Our graph incorporates the order on operations given by the program text, enabling us to do without locks even when database conflict graphs would suggest that locks are necessary.

1. Introduction

Programs on shared memory MIMD computers, e.g. the NYU Ultracomputer [GGK] or the IBM RP3 machine [PBG], consist of many *program segments* each executed sequentially by a single processor. The memory locations accessed by these programs are either locations in shared memory modules, or local memory locations, which we call *registers*. An *operation* in a given processor follows one of four basic patterns:

- (1) read from registers (in that processor) and write to registers;
- (2) read from registers (in that processor) and write to a single location in a shared memory module;
- (3) read a single location from a shared memory module and write to registers (in that processor);
- (4) force a constant to either a register (in that processor) or a shared memory module location.

We call a register or a location in a memory module by the generic name *variable*. Thus, an operation consists of a read of zero or more variables followed by a write to one or more variables. There are two restrictions: an operation accesses only one shared

memory module location and an operation accesses registers only in the processor in which it is issued.¹

Different program segments may execute at vastly different speeds, without violating any reasonable criterion of correctness. Thus, the only semantic correctness requirement for such an execution is embodied in the following principle [La, LF].

Sequential Consistency

The outcome of a sequentially consistent computation (or execution) is as if all the operations were executed in some sequential order, with the constraint that the operations of each individual processor appear in this sequence in the order they were executed by the processor.² That is, the outcome must be the same as it would be in some interleaving of the operations such that if u precedes v in a serial program, then u precedes v in the interleaving.

The computer design helps us achieve sequential consistency by ensuring the *atomicity* of the execution of memory accesses: If two operations access the same memory location the two accesses will have the same effect as if they executed serially.

A technique for achieving sequential consistency is given by Lamport in [La]: Requests for memory access are issued by each processor in the order they occur in its program segment; and requests are serviced by variables in the order they arrive. This solution is correct provided that requests arrive to memory in the order they are issued. If requests may arrive out of order then the solution fails.

It is easy to enforce correct arrival order for memory requests: when executing each sequential program, wait for each memory access to complete (and return a value or an acknowledgement) before issuing the next one. Whatever execution finally results must be consistent with the order within each sequential program.

In the context of our machine description, such a solution results in severe loss of efficiency. The network that connects processors to shared memory is likely to have a large latency. On the other hand, if the network is message switched, then one memory

¹ The results are valid for operations that access more than one shared memory location, provided that shared memory access is atomic.

² The semantics of a sequence generalizes the semantics of individual operations in the natural way. Each operation maps a state to a state. In a sequence, the $i+1$ st operation produces a new state based on the state

request may be issued before a previous one has completed. For example a processor that accesses two different memory locations may not need to delay one access until the other one completes. This allows the network to attain a high throughput, despite the large latency.

Unfortunately, we can not detect possible violations of sequential consistency by analyzing each program segment in isolation. Consider, for example the following two program segments (the example is taken from [Co]).

<i>Segment 1</i>	<i>Segment 2</i>
$p_1 \ X := 1;$	$q_1 \ y := Y;$
$p_2 \ Y := 1;$	$q_2 \ x := X;$

X and Y are module memory locations shared by both program segments and x and y are different registers. Assume that initially $X=Y=0$. No interleaving of these operations consistent with the order in each program can lead to a state where $x=0$ and $y=1$. Indeed, if $x=0$ then operation q_2 was executed before p_1 ; but then q_1 should take effect before p_2 , so that $y=0$. If the accesses to shared memory in either the first program segment or in the second program segment are executed out of order, then it is quite possible to obtain this inconsistent result (figure 1).

Note, however, that this is the only impossible result. For example, the access pattern q_1, p_2, p_1, q_2 , would yield $x=1$ and $y=0$. This would be entirely acceptable, since it produces the same results as the interleaving q_1, p_1, p_2, q_2 ³.

This example has shown that it may be necessary for a processor to ensure that certain operations are executed in the order they appear in a program segment (even when there are no data dependencies within that segment). We assume that a processor may delay one memory access until acknowledgment of some previous memory request is received. This is the only mechanism we allow: there is no inter-segment communication other than through shared variables.

Conflicts may occur only when shared variables are modified. Thus, accesses to private variables (i.e. variables that occur in one program segment only) cannot cause

produced by the i th operation.

³ The reader may observe that the sequential consistency requirement is much weaker than the serialization principle addressed in database theory [KP], which would forbid this second outcome.

problems. The same is true for accesses to shared read-only variables. Private and read-only variables are easy to detect. The following simple policy guarantees correct execution of parallel code (see [EGK]):

When an access to a shared memory location is issued, wait for acknowledgement if a shared read-write variable is accessed; otherwise issue the next operation immediately.

We show in this paper that it is possible to improve on this simple policy by doing a more elaborate analysis of data dependencies. We give a characterization of the minimal set of delays that must be introduced in the code in order to enforce sequential consistency.

We have addressed so far the problem of serializing the execution of atomic machine operations. This is the issue facing a machine designer. A high level language designer or a user face a different issue: for them the sequential consistency principle should apply to those operations in the high level language that are defined to be atomic. (This principle turns out to be far closer to the principle of serializability for database systems.)

When high level language constructs are compiled into machine code, one high level atomic operation translates into possibly several machine operations. Each such block of code has to be executed atomically. Delays are not sufficient to enforce such atomicity. In the general case, when one "atomic" block of code accesses several memory locations, it might be necessary to lock those locations in order to preserve atomicity.

A straightforward solution is to lock all locations accessed by the operations within an atomic block before one starts executing operations within the block, and release the lock after all operations have executed. Locking may be an expensive operation and reduces concurrency. We want to avoid the need for locking to the greatest possible extent.

We can improve on the straightforward solution by doing a more refined data dependency analysis. The analysis detects when delays are sufficient to enforce correct execution; it determines a minimal set of locks that must be used.

There is a significant similarity between our analysis techniques and the "conflict graph analysis" techniques pioneered by the distributed database project SDD-1 [BSR, BG] for concurrency control. A conflict graph is an undirected graph that indicates potential conflicts between "transactions" (where transactions are analogous to program segments) in different classes, where the class of a transaction consists of the variables it reads and

the variables it writes. Different considerations make the analyses diverge from this point: the correctness criteria of the analyses differ (see previous footnote); and the intended order of operations within a program is known in advance in our case and we use it in our analysis, but this ordering is ignored by conflict graph analysis. There are other examples of the interplay between database concurrency control and synchronous parallel algorithms, e.g. [UW]. These merit further exploration.

2. Main Result

2.1. Preliminaries

We consider a concurrent execution of a parallel program consisting of several straight line program segments. (Later, we will generalize these results.) As noted above, we assume that every operation accesses at most one location in the memory modules. An operation that modifies a memory location is a *write*; an operation that accesses a memory location, but does not modify its contents is a *read*. The access to each variable is *atomic*, (i.e. two accesses to the same variable behave as if they occur serially in some order). However, since an operation may access several variables, all those accesses together may not necessarily be atomic.

An execution of the program is characterized by two relations: the order in which the operations are written in the program and the order in which they access variables. Execution order is relevant only for operations that *conflict*, i.e. operations that access a common variable. The execution is sequentially consistent if each operation behaves atomically and the conflict and issuing orders are consistent. The execution order is not known in advance; however, we do know in advance what (unordered) pairs of operations conflict. The execution order should define an ordering on each such pair.

Let us formalize the previous discussion. Two relations R and S are *consistent* if $R \cup S$ can be extended to a total ordering (a total ordering is a partial ordering that orders every two elements). A relation can be extended to a total ordering iff its transitive closure is irreflexive. Thus R is consistent with S iff the graph of the relation $R \cup S$ has no cycles.

Let R be a symmetric relation. The relation O is an *orientation* of R if whenever uRv then either uOv or vOu holds. The relation O is a *proper orientation* of R if O is an acyclic

orientation of R . If O is a proper orientation of R and uRv then exactly one of uOv or vOu holds. O is obtained from R by orienting the edges of the (undirected) graph representing R so as to obtain a directed acyclic graph.

We characterize the code by a relation $\langle V, P \rangle$. V is the set of operations in the program. Intuitively, uPv if u occurs before v in one of the program segments. Here, P consists of a union of disjoint serial orders, one per program segment;

Two distinct operations are said to *conflict* if they access one or more of the same variables and at least one writes one of these variables.⁴ We denote the conflict relation by C . The relation C is symmetric; it is not transitive or reflexive.

An *execution* E is an orientation of the conflict relation C . Any execution of a program defines an execution relation E : uEv if there is a variable that is accessed by u before it is accessed by v , such that either u or v writes on this variable.

An execution E is *correct* if it is acyclic and consistent with P , i.e. if $E \cup P$ can be extended to a total ordering.

An execution E is correct iff E is a proper orientation of C and the graph of $P \cup E$ has no cycles. Informally, E is correct if there is a total ordering of the operations that is consistent with the order within each process (i.e. P), and with the order in which memory accesses take effect (i.e. E). Thus, if E is correct in an execution, then the execution is sequentially consistent.

In the example above, if q_1Ep_2 and p_1Eq_2 are the only pairs in E , then the graph $E \cup P$ has no cycles. One possible ordering is q_1, p_1, p_2, q_2 . However if p_2Eq_1 and q_2Ep_1 , there is a cycle in $E \cup P$ as follows: $p_2Eq_1Pq_2Ep_1Pp_2$. The cycle indicates that this second execution order is not correct.

One can control the order of execution of operations by introducing *delays*. Delays are introduced between operations executed by the same processor. We denote by uDv the fact that operation v is delayed until operation u is executed (the processor does not start memory accesses on behalf of v until all memory accesses on behalf of u have completed).

⁴ Actually, one can use a slightly more general definition: operations o and o' conflict if either there is some state s such that the state obtained by applying o to s and then o' to the resulting state differs from the state obtained by applying o' then o . Under such a definition, two additions to a location would not conflict. We use the less general definition for simplicity of presentation.

The relation D is a partial order.

Delays can be used to sequentially consistent execution of serial code. We require that D orders every conflicting pair of operations within a program segment. All the delay predicate we propose below have this property provided that P orders every pair of conflicting relations within the same program segment.

If u and v are conflicting instructions within a program segment then either uDv or vDu , so that either all the memory accesses of u are executed before any memory access of v or all memory accesses of v are executed before any memory access of u . Thus, if u and v belong to the same program segment and uEv then uDv .

The relation $E \cup D$ cannot have a cycle that is wholly contained within one program segment; such cycle would also be a cycle of D .

If uDv , or uEv and u and v belongs to the same program segment, then all the memory accesses of u occur before any memory access of v . If uEv , and u and v belong to distinct program segments, then the (unique) access of u to shared memory occurs before the (unique) access of v to shared memory. Thus if u and v are two operations that access shared memory, and there is a path from u to v in the graph of $E \cup D$, then the access of u to shared memory occurs before the access of v . It follows that the relation $E \cup D$ cannot have a cycle that includes operations that access shared memory.

We have shown that $E \cup D$ is acyclic; E is a partial order consistent with D . Conversely, any execution order E that is consistent with the delay relation D may arise from the execution of the program with these delays.

We are looking for a delay relation D with the following properties:

- 1) Any execution order that is consistent with D is correct, i.e. ensures that E is a proper orientation of C and that E is consistent with P .
- 2) No proper subset of D has property 1.

We say that D *forces consistency* if it has the above property 1. For example, the relation P forces consistency for serial program segments. This amounts to forcing serial execution of memory accesses within each program. In many practical cases P is not minimal, so violates property 2.

2.2. Border pairs

An execution E is incorrect if $P \cup E$ has cycles. All such cycles are also cycles of the graph of $P \cup C$. This graph is known in advance, and its cycles indicate potential violations of sequential consistency at run time. Suppose that we enforce a delay uDv for every pair of operations uv such that uPv , and uv is an edge on a cycle of $P \cup C$. Then each cycle of $P \cup C$ is also a cycle of $D \cup C$. It follows that each cycle of $P \cup E$ would be a cycle of $D \cup E$. Since $D \cup E$ is acyclic, $P \cup E$ is acyclic, and E is correct.

It is not necessary to enforce delays for each P edge on a cycle of $P \cup C$: If $uPvPw$ are consecutive P edges on a cycle of $P \cup C$, then a delay between u and w is sufficient to break this potential cycle. This is formalized below.

A (cyclic) list of instructions $\sigma = (v_0, \dots, v_{n-1}, v_0)$ is a *mixed cycle* of (P, C) if σ is a cycle in the graph of $P \cup C$ that contains edges both from P and from C . (Note that there may be both a P edge and a C edge between v_i and v_{i+1} . So the cycle may be fully contained in edges from C . But then some consecutive pair v_i and v_{i+1} in the cycle must also be a P edge.)

The ordered pair of operations $v_i v_{i+1}$ is an (ordered) *border pair* of the mixed cycle σ if (all addition is modulo n)

- (1) $v_{i-1} C v_i$,
- (2) $v_i P v_{i+1}$, and
- (3) $v_{i+1} C v_{i+2}$.

In the example above $(p_1 p_2 q_1 q_2 p_1)$ is a mixed cycle, and $p_1 p_2$ is a border pair of this cycle.

Let uBv be the relation “ uv is a border pair of a mixed cycle”. B is a transitive relation. Indeed, let $u_0 u_1$ be a border pair in the mixed cycle $(u_0, u_1, v_2, \dots, v_{m-1}, u_0)$ and let $u_1 u_2$ be a border pair in the mixed cycle $(u_1, u_2, v'_3, \dots, v'_{n-1}, u_1)$. Then

$$(u_0, u_2, v'_3, \dots, v'_{n-1}, u_1, v_2, \dots, v_{m-1}, u_0)$$

is a mixed cycle, and $u_0 u_2$ is a border pair of this cycle.

It is also easy to see that if v_0v_1 is a border pair of a mixed cycle, then it is also a border pair of a simple mixed cycle.

Delay Condition: If v_0Bv_1 then wait for v_0 to complete before issuing v_1 , i.e. ensure v_0Dv_1 .

Theorem 1: B forces consistency.⁵ That is, if an execution satisfies the delay condition, it will be sequentially consistent.

Proof: Let E be an incorrect execution order. Let $(v_0, v_1, \dots, v_{n-1}, v_0)$ be a simple cycle in the graph of $P \cup E$ ($n > 1$). Since both P and E are irreflexive and asymmetric, the cycle contains edges from both P and from E . By transitivity of P , we may assume w.l.o.g. that the cycle does not contain two successive P related pairs.

Since $E \subseteq C$ if v_iEv_{i+1} , then v_iCv_{i+1} . It follows that $(v_0, \dots, v_{n-1}, v_0)$ is a mixed cycle of (P, C) . But then each ordered pair $v_i v_{i+1}$ such that v_iPv_{i+1} is a border pair, so that v_iBv_{i+1} . Therefore $(v_0, \dots, v_{n-1}, v_0)$ is a cycle of $B \cup E$, and E is not consistent with the delay relation. So $P \cup E$ contains no cycle. $\square$ •

2.3. Examples

The following examples illustrate the last results. These examples only show conflicts at memory module locations. We assume that the various operations of the same program segment don't conflict in the registers of the processor executing that segment.

The first example illustrates that data dependencies within one program segment are handled correctly as a particular case of border pairs.

Program

- (a) Read A
- (b) Write B
- (b') Read B
- (a') Write A

We have a mixed cycle (aa'a) and a mixed cycle (bb'b). The pairs aa' and bb' are border pairs, so that (a') is delayed until (a) terminates, and (b') is delayed until (b)

terminates.

The next example is more complex.

<i>Segment 1</i>	<i>Segment 2</i>	<i>Segment 3</i>
(a) Read A	(c') Write C	(d'') Read D
(b) Read B	(b') Write B	(a'') Write A
(c) Read C	(d') Write D	

We have three mixed cycles: (abb'd'd''a'a), (acc'd'd''a'a), and (bcc'b'b). See figure 2. The border pairs in the first program segment are ab, ac, and bc; in the second program segment they are c'b', c'd', and b'd'; in the third program segment d'a'' is a border pair. The pair ac is not needed as it is implied by ab and bc; same goes for c'd'. The delays needed are ab, bc, c'b', b'd' and d'a''. In this example each memory access is delayed until the previous access has terminated.

This example shows that it is not necessary to record all delays implied by the border pair relation *B*; it is sufficient to record a set of delays such that their transitive closure contains *B*. We shall return to this later.

The last example illustrates that our method can save delays.

<i>Segment 1</i>	<i>Segment 2</i>
(a) Read A	(b') Write B
(b) Read B	(c') Write C
(c) Read C	(d') Write D
(d) Read D	(a') Write A

Figure 3 shows the conflict and program order edges. We have four mixed cycles: (abb'c'cdd'a'a), (abb'a'a), (acc'a'a) and (add'a'a). See figure 4. The border pairs in the first program segment are ab, ac, ad, and cd; in the second program segment they are b'c', b'a', c'a', and d'a'. In the first program segment all operations are delayed until (a) is executed and (d) is delayed until (c) is executed. In the second program segment (a') is delayed until all operations have been executed; (c') is delayed until (b') has been executed. On the other hand, (b) can be executed after (c) or (d), even though these are

⁵ Remember, we are dealing with straight line code.

accesses to shared read-write variables. A symmetric condition holds for the second program segment.

2.4. Minimal delays

The last example also shows that our solution is not minimal: in order to force consistency it is sufficient to delay all the operations in the first program segment until (a) has terminated, and to delay the execution of (a') in the second program segment until all the other operations have terminated. The delays between (c) and (d) in the first program segment, and between (b') (c') in the second program segment are superfluous.

It is quite easy to see what went wrong in the last example: Although (cdd'a'abb'c') is a mixed cycle, it is not a *minimal mixed cycle*; the mixed cycles (abb'a'), (acc'a') and (add'a') are all contained within this cycle. This is repaired by using minimal cycles in the definition of delay pairs.

A simple mixed cycle $(v_0, \dots, v_{n-1}, v_0)$ of (P, C) is *minimal* if either no proper subset of its nodes form a mixed cycle, or any proper subset that does consists of consecutive nodes (v_i, v_{i+1}, v_i) where $v_i P v_{i+1} C v_i$ (addition is modulo n , as usual).⁶ See figure 5.

Intuitively, a minimal mixed cycle should have at most two operations per program segment that are related by the program ordering P (note that if the program ordering is total, there is no need for the last restriction). If there were three, the middle one could be eliminated. The two operations from the same program segment should also be consecutive in the cycle, i.e. one should be v_i and the other should be v_{i+1} for some i . If there are two operations o and o' from the same program segment such that $o P o'$ but o and o' are not consecutive in the cycle, then the path from o to o' can be removed to obtain a smaller cycle.

Lemma 1: Let Q and C be relations such that Q is transitive and C is symmetric. Let $\sigma = (v_0, \dots, v_{n-1})$ be a simple mixed cycle of (Q, C) . Then σ is a minimal iff the following conditions hold.

⁶ When memory access operations are just reads and writes on single memory locations (as opposed to sets of locations), then the definition of a minimal cycle can be simplified: A mixed cycle is minimal if no proper subset of its nodes forms a mixed cycle. The more general definition makes no assumptions on the semantics of memory accesses.

(1) $v_i Q v_j$ and $i \neq j$ implies $j = i \pm 1 \pmod{n}$.

(2) $v_i C v_j$ and $i \neq j$ implies $j = i \pm 1 \pmod{n}$.

Proof: Let $\sigma = (v_0, \dots, v_{n-1}, v_0)$ be a minimal mixed cycle.

(only if) Assume $v_i Q v_j$, where $j \neq i$ and $j \neq i \pm 1$.⁷ Then

$$\pi = (v_j, v_{j+1}, \dots, v_i, v_j)$$

is a cycle in $Q \cup C$ consisting of a proper subset of nodes of σ . Since σ is minimal it follows that π is not mixed, so that the path $v_j, v_{j+1}, \dots, v_i$ is contained in $Q - C$. Since Q is transitive we obtain that $v_j Q v_i$. But then $(v_i, v_{i+1}, \dots, v_j, v_i)$ is a cycle, and this cycle is mixed. Thus σ is not minimal.

Assume $v_i C v_j$, where $j \neq i$ and $j \neq i \pm 1$. Then both $(v_i, v_{i+1}, \dots, v_j, v_i)$ and $(v_j, v_{j+1}, \dots, v_i, v_j)$ are cycles in $Q \cup C$; at least one of these two cycles is mixed. Thus σ is not minimal. It follows that if σ is minimal then conditions (1) and (2) hold true.

(if) If σ is a simple cycle that fulfills these two conditions, then no edge connects pairs of nodes on σ in the graph of $Q \cup C$, with the exception of the edges that are on the cycle, or their reversals. It follows that any cycle in $Q \cup C$ with nodes in the set $\{v_0, \dots, v_{n-1}\}$ is either identical to σ or the reversal of σ , or is of the form (v_i, v_{i+1}, v_i) . Thus, σ is minimal. $\square$

Lemma 2: Let Q be a transitive relation, C be a symmetric relation, and E be a proper orientation of C . Let $\sigma = (v_0, \dots, v_{n-1}, v_0)$ be a simple mixed cycle of (Q, E) . Then the following conditions are equivalent.

(1) σ is a minimal mixed cycle of (Q, E) .

(2) σ is a minimal mixed cycle of (Q, C) .

(3) σ fulfills conditions (1) and (2) of the previous lemma, i.e. if $v_i Q v_j$, with $i \neq j$, then $j = i \pm 1$, and if $v_i C v_j$, with $i \neq j$, then $j = i \pm 1$.

Proof: We have proven in the previous lemma the equivalence of conditions (2) and (3). Clearly, if σ is a simple mixed cycle of (Q, E) , and is a minimal mixed cycle of (Q, C) , then it is also a minimal mixed cycle of (Q, E) . Thus (2) $\Rightarrow$ (1). We shall complete the proof

⁷ It is irrelevant to our proof whether $i < j$ or $i > j$.

by showing that (1) $\Rightarrow$ (3).

If $v_i Q v_j$, where $j \neq i$ and $j \neq i \pm 1$ then the argument used in the proof of the previous lemma shows that σ is not minimal.

If $v_i C v_j$, where $j \neq i$ and $j \neq i \pm 1$ then either $v_i E v_j$ or $v_j E v_i$. Assume w.l.o.g. that the former holds. (Again, we don't care whether $i < j$ or $i > j$.) Then

$$\pi = (v_j, v_{j+1}, \dots, v_i, v_j)$$

is a cycle in the graph of $Q \cup E$. Since E is acyclic the path $v_j, v_{j+1}, \dots, v_i$ must contain an edge in $Q - E$; the cycle π is mixed. $\square$

An ordered pair $v_0 v_1$ is a *critical pair* if it is a P edge of a minimal mixed cycle of (P, C) .

Theorem 2: Let D be the “critical pair” relation. Then D forces consistency.

Proof: Assume that D does not force consistency. Let E be an execution order relation that is consistent with D , but not with P . Let $\sigma = (v_0, \dots, v_{n-1}, v_0)$ be a minimal cycle in $P \cup E$. Any minimal cycle in $P \cup E$ must be mixed since both P and E are irreflexive and transitive. Thus σ is a minimal mixed cycle of (P, E) and, according to Lemma 2, σ is a minimal mixed cycle of (P, C) . Thus, if $v_i P v_{i+1}$ then $v_i D v_{i+1}$, and σ is a cycle in the graph of $E \cup D$; a contradiction. $\square$

The last solution turns out to be optimal.

Theorem 3: Let R be a relation that is consistent with P and forces consistency. Then the critical pair relation D is contained in the transitive closure of R .

We shall use the following lemma in our proof:

Lemma 3: Let $G = \langle V, E \rangle$ be a directed acyclic graph, and let u and v be two nonadjacent nodes of G . Then either $G_1 = \langle V, E \cup uv \rangle$ or $G_2 = \langle V, E \cup vu \rangle$ are acyclic graphs.

Proof: If both G_1 and G_2 contain cycles, then there is a path in G from u to v , and a path from v to u . Hence G contains a cycle; contradiction. $\square$

Proof of Theorem 3: We assume w.l.o.g. that R is transitive; we shall prove that $D \subseteq R$. Let $\sigma = (v_0, \dots, v_{n-1}, v_0)$ be a minimal mixed cycle (in (P, C)), and let $v_0 v_1$ be a critical pair in this cycle. Assume that $v_0 v_1 \notin R$. We shall construct an execution order E that is consistent with R , such that σ is a cycle in $E \cup P$. Hence E is not correct, and R does not

force consistency.

Define the relation E' by $v_i E' v_{i+1}$ if $v_i C v_{i+1}$. (We will later extend E' to an incorrect conflict ordering E .)

Claim: The graph of $E' \cup R$ has no cycles.

The claim is trivial if $\sigma = (v_0, v_1, v_0)$: Then $E' = v_1 v_0$ and R contains no path from v_0 to v_1 , otherwise $v_0 v_1 \in R$, contradicting our assumption.

Consider now the case $n > 2$. Assume that $E' \cup R$ has a cycle; let π be a minimal such cycle. This cycle fulfills conditions (1) and (2) of Lemma 1 (by Lemma 2). In particular, π does not contain two successive R edges, so that all the nodes of π are endpoints of edges in E' . It follows from our construction of E' that the nodes of π are nodes of σ , and by the minimality of σ , π and σ have the same set of nodes. As σ fulfills conditions (1) and (2) of Lemma 1 the edges of π must coincide with edges of σ , or their reversal. In particular, either $v_0 v_1$ or $v_1 v_0$ is an edge of π . But $v_1 v_0 \notin R$ (since $v_0 v_1 \in P$), and $v_0 v_1 \notin C$ (otherwise (v_0, v_1, v_0) is a mixed cycle of (P, C)). It follows that $v_0 v_1 \in R$, contradicting our assumption. This proves the claim.

As $E' \cup R$ is acyclic, we can extend E' to an execution order E consistent with R by repeated applications of Lemma 3. However $E \cup P$ contains a cycle - the cycle σ ; hence E is not consistent with P and is therefore incorrect. $\square$

3. Large Atomic Operations And Locks

In this section, we consider the problem of enforcing sequential consistency at the level of complex atomic constructs, i.e. instructions in the source program that translate to multiple machine language operations. We assume that two mechanisms can be used for that purpose:

- (i) A processor may delay issuing an operation until some other operation has terminated (these are the delays used in the previous sections).
- (ii) A processor may *lock* a memory location. There are two kinds of locks: read locks and write locks. Several processors may simultaneously hold read locks on the same location; only one processor may hold a write lock. Moreover, if one processor holds a write lock on a location, no other processor may hold a read lock on that location. A processor may read a location only if it holds a read or a write lock on it; it may

modify a location only if it holds a write lock on it. Sometime after the processor has finished accessing a location, it *unlocks* the location.

We shall restrict our attention to nonbranching code. The results can be extended to branching code as we will see in the next section. The problem is formalized as follows. We have two relations defined on the set V of operations:

- (i) The relation P specifies the order within each program segment: uPv if u and v are operations of the same program segment, and u precedes v .
- (ii) The relation A specifies atomicity constraints: uAv if u and v are operations of the same program segment, and belong both to the same atomic block.

The following conditions (1)-(4) state that the order of execution of atomic blocks is acyclic, and the order of execution of operations within each atomic block is acyclic.

- (1) The relation A is an equivalence relation.
- (2) The relation P is a partial order (i.e. transitive and irreflexive). Hence, so is $P \cap A$.
- (3) Let P/A be the relation induced on the equivalence classes of A by the relation P : $[u]P/A[v]$ if there exist $u' \in [u]$ and $v' \in [v]$ such that $u'Pv'$. Then P/A is a partial order.
- (4) The relation P is “closed” under A ; i.e. if $[u]P/A[v]$ then uPv .

These conditions imply an alternate point of view: there is an order relation P/A defined on A equivalence classes, and an order relation $P \cap A$ defined within equivalence classes. The relation P is the lexicographic ordering defined by P/A and $P \cap A$.

The conflict relation C is defined as originally, i.e. uCv if both access the same location and at least one is a write.

A total ordering S of the code is *correct* if

- (i) S is consistent with P , i.e. $P \subseteq S$, and
- (ii) Operations belonging to the same equivalence class of A are executed consecutively, i.e. if uSv , vSw , and uAw , then uAv (and vAw).

A execution order E is *correct* if there is an extension of E to a correct total ordering.

Condition (ii) implies that the effect of the execution is as if each atomic block were executed indivisibly; the first condition implies that the effect of the execution is as if the operations were executed in the order specified by P .

Let L be an equivalence relation and D be a partial order both defined on pairs of operations that belong to the same program segment. The relation L is a *locking relation* that partitions the operations into equivalence classes (*locking sets*) that are locked together; D is a delay relation. Execution of the program with the constraints determined by L and D is defined as follows.

Delay and locking conditions: Each processor issues operations while respecting the following two restrictions:

- (1) If uDv then v is not issued until u has executed.
- (2) Let $[u]$ be the equivalence class of operation u under the equivalence relation L . Then, before issuing the first operation from $[u]$, a processor read-locks all memory locations read but not written by operations from $[u]$; it write-locks all locations written by operations from $[u]$; it releases these locks after executing all operations in $[u]$.

An execution order E is *consistent* with L and D if it may result from an execution that fulfills the delay and locking conditions defined by D and L . The pair of relations (L, D) *forces correctness* if any execution order that is consistent with D and L is correct. For example (A, P) forces correctness. We show in this section how to improve on this trivial solution.

Lemma 4: The following assertions are equivalent:

- (1) The execution order relation E is correct.
- (2) $P \cup E$ and $P/A \cup E/A$ are partial orders.
- (3) $(P \cap A, E \cap A)$ and $(P \cup A, E - A)$ have no mixed cycles

Proof: Note first that the first conjuncts of conditions (2) and (3) are not redundant: Even if $P/A \cup E/A$ is a partial order, $P \cup E$ might still have cycles that are wholly contained within an equivalence class of A . Similarly, $(P \cap A, E \cap A)$ might have a mixed cycle that is not a mixed cycle of $(P \cup A, E)$, as it is contained within a single A equivalence class.

If $P \cup E$ is not a partial order then it has a cycle σ . Since P and E are acyclic, σ contains edges in $P - E$ and edges in $E - P$. If the nodes of σ are contained in an equivalence class of A , then it is a mixed cycle of $(P \cap A, E \cap A)$. Otherwise this is a cycle of $(P \cup A, E - A)$. Since P/A is acyclic, the cycle contains edges from $E - A$, and is mixed.

If $P/A \cup E/A$ is not a partial order then it has a cycle. This cycle corresponds to a mixed cycle of $(P \cup A, E - A)$.

Thus, $(3) \Rightarrow (2)$ (we have shown the contrapositive).

If (2) holds then it is possible to extend $P/A \cup E/A$ to a total order of the equivalence classes of A ; it is possible to extend $P \cup E$ to a relation that totally orders each equivalence class of A . The lexicographic ordering defined on the nodes of the program by these two order relations is a correct extension of E . Thus $(2) \Rightarrow (1)$.

Finally, it is easy to see that if E can be extended to a correct total ordering, then $(P \cap A, E \cap A)$ and $(P \cup A, E - A)$ have no mixed cycles, so that $(1) \Rightarrow (3)$. $\square$

Lemma 4 indicates the way to enforce correct execution. One can prevent the occurrence of cycles in $P \cup E$ within equivalence classes, by delaying critical pairs in minimal mixed cycles of $(P \cap A, C \cap A)$. Similarly, one can prevent the occurrence of mixed cycles in $(P \cup A, E - A)$, by delaying critical pairs in minimal mixed cycles in $(P \cup A, C - A)$. However, as $P \cup A$ is not irreflexive, we may get contradictory requirements: i.e. u should be delayed until v executes, and vice versa. In such cases, we must lock the variables accessed by u and v before either executes. Once we have found the set of operations that must be locked, we may relax the delay requirements, as locks take care of part of the constraints we intended to enforce with delays.

Let us give now a formal description. Let $D_{\cap}$ be the set of critical pairs in minimal mixed cycles of $(P \cap A, C \cap A)$ Let $D_{\cup}$ be the set of critical pairs in minimal mixed cycles of $(P \cup A, C - A)$.

Let $L \subseteq A$ be an equivalence relation. Define $D_{int}(L)$ to be the set of critical pairs in minimal mixed cycles of $(P \cap L, C \cap L)$ (i.e. cycles whose nodes are contained within an equivalence class of L).

Finally, let

$$D(L) = D_{int}(L) \cup ((D_{\cap} \cup D_{\cup}) - L) .$$

We have

Lemma 5: Any execution order E that is consistent with L and $D(L)$ is correct.

Proof: Let E be an execution order that results from an execution with delay and locking conditions $D(L)$ and L . Assume, by contradiction, that E is not correct.

Case a: $(P \cap A, E \cap A)$ has a mixed cycle. Let $\sigma = (v_0, \dots, v_{n-1})$ be a minimal such cycle. According to Lemma 1 σ is a minimal mixed cycle of $(P \cap A, C \cap A)$.

a.1 - The nodes of σ are contained in an equivalence class of L . Then σ is minimal mixed cycle of $(P \cap L, C \cap L)$; $v_i P v_{i+1}$ implies $v_i D_{int}(L) v_{i+1}$, so that σ is a cycle in $D_{int}(L) \cup E$; a contradiction.

a.2 - The nodes of σ are not all within the same locking set. If $v_i A v_{i+1}$ and $v_{i+1} P v_{i+2}$ then $v_i P v_{i+2}$. It follows that σ does not contain A edges followed by P edges. The cycle σ can be split into disjoint segments

$$v_{i_0}, \dots, v_{i_1-1}, v_{i_1}, \dots, v_{i_2-1}, \dots$$

such that the nodes $v_{i_j}, \dots, v_{i_{j+1}-1}$ belong to the same equivalence class of L . There is more than one such segment. If $v_{i_{j-1}} E v_{i_j}$ then v_{i_j} is executed after $v_{i_{j-1}}$; the instructions $v_{i_{j-1}}$ and v_{i_j} conflict, and require conflicting locks on the same location. The lock for v_{i_j} is secured only after the lock for $v_{i_{j-1}}$ is released and, therefore, only after all the operations in the locking set of $v_{i_{j-1}}$ have executed. Thus v_{i_j} executes after $v_{i_{j-1}}, \dots, v_{i_j-1}$. If $v_{i_{j-1}} P v_{i_j}$ then $v_{i_{j-1}} (D \cap L) v_{i_j}$ and, again, v_{i_j} is executed after $v_{i_{j-1}}$. Also, by a previous remark, $v_{i_{j-1}} = v_{i_{j-1}}$, i.e. the segment of consecutive operations in the same lock set as $v_{i_{j-1}}$ has length one. In either cases $v_{i_{j-1}}$ is executed before v_{i_j} yielding a contradiction.

Case b: $(P \cup A, E - A)$ has a mixed cycle. Let $\sigma = (v_0, \dots, v_{n-1})$ be a minimal such cycle. Then σ is a minimal mixed cycle of $(P \cup A, C - A)$. If $v_i (P \cup A) v_{i+1}$ then either $v_i L v_{i+1}$ or $v_i (D \cup L) v_{i+1}$. The same argument as in case a is used to derive a contradiction. $\square$

The claim of the last theorem may be vacuous: Since $P \cup A$ has cycles $D \cup L$ may turn out to be cyclic too. In this case the set of delays $D(L)$ is unenforceable, and no execution order is consistent with L and $D(L)$. Conversely, if $D(L) \cap L$ and $D(L)/L$ are both acyclic then the delay and locking conditions represented by $D(L)$ and L are

enforceable: $D(L)$ is acyclic, and there is no cyclic dependency of locking operations. Our purpose is to find the “smallest” locking relation L such that $D(L)$ is acyclic.

Let $D = D_{\cap} \cup D_{\cup}$. Then $D(L) = D_{int}(L) \cup (D - L) \subseteq L \cup (D - L)$. If both uv and vu are in the transitive closure of D then the delays uDv and vDu are unenforceable; we must put the pair uv in L . It turns out that it is also sufficient to put these pairs into L .

Let us call pairs uv such both uv and vu are in the transitive closure of D *tight*. Let T be the tight relation. T is an equivalence relation; the equivalence classes of T are the strongly connected components of the graph of D .

Lemma 6: Let T be the tight equivalence relation. Let $D(T)$ be the delay relation associated with the locking relation T . Then $D(T) \cap T$ and $D(T)/T$ are acyclic.

Proof: We have $D(T) \cap T = D_{int}(T) \subseteq P$, and P is acyclic. Also, $D(T) - T = D - T$, so that $D(L)/L = D/L$. But D/L is the graph induced by D on the strongly connected components of D ; this graph is acyclic. $\square$

The following theorem sums up the results of this section.

Theorem 4: Let $D_{\cap}$ be the set of critical pairs of minimal mixed cycles of $(P \cap A, C \cap A)$. Let $D_{\cup}$ be the set of critical pairs of minimal mixed cycles of $(P \cup A, C - A)$. Let T be the equivalence relation consisting of the set of pairs uv such that both uv and vu are in the transitive closure of $D_{\cap} \cup D_{\cup}$. Let $D_{int}(T)$ be the set of border pairs of minimal mixed cycles of $(P \cap T, C \cap T)$. Finally, let

$$D(T) = D_{int}(T) \cup ((D_{\cap} \cup D_{\cup}) - T) .$$

Then the locking condition T and delay condition $D(T)$ are enforceable, and force correct execution. $\square$

3.1. Examples

We illustrate the last theorem with several examples. As before, these examples only show conflicts at memory module locations. We assume that the various operations of the same program segment don't conflict in the registers of the processor executing that segment. We enclose blocks of statements that are to be executed atomically in curly

brackets; a comma separates statements that can be executed concurrently (semicolon has higher priority than comma).

First Example

<i>Segment 1</i>	<i>Segment 2</i>
$\{p_1 \text{ Write } a;$	$q_1 \text{ Read } a;$
$p_2 \text{ Write } b,$	$q_2 \text{ Read } b$
$p_3 \text{ Read } b;$	
$p_4 \text{ Read } a\}$	

$(P \cap A, C \cap A)$ has a minimal mixed cycle $(p_1 p_2 p_3 p_4 p_1)$. See figure 6. The pairs $p_1 p_2$ and $p_3 p_4$ are critical pairs of this cycle. $(P \cup A, C - A)$ has a minimal mixed cycle $p_2 p_1 q_1 q_2 p_2$. The pairs $p_2 p_1$ and $q_1 q_2$ are critical pairs of this cycle. Thus, the pair $p_1 p_2$ is tight and a and b must be locked together when these two instructions are executed. All other accesses are executed separately. p_4 is delayed until p_3 executes and q_2 is delayed until q_1 executes. No delay is required between p_2 and p_3 even though these operations are on a cycle: the lock ensures that either p_3 occurs before p_1 and p_2 or vice versa.

Second Example

<i>Segment 1</i>	<i>Segment 2</i>	<i>Segment 3</i>
$p_1 \text{ Read } a;$	$\{q_1 \text{ Write } a;$	$r_1 \text{ Read } c;$
$p_2 \text{ Read } b$	$q_2 \text{ Write } c\}$	$r_2 \text{ Write } b$

There are no cycles, neither in the graph of $P \cap A \cup C \cap A$, nor in the graph of $P \cup A \cup C - A$. See figure 7. Thus no delays or locks are needed for the correct execution of this program. Any interleaving will be equivalent to some interleaving where q_1 and q_2 are executed consecutively.

Third Example

<i>Segment 1</i>	<i>Segment 2</i>	<i>Segment 3</i>
------------------	------------------	------------------

p_1 Read a;	$\{q_1$ Write a;	r_1 Write b;
p_2 Read b	q_2 Write c}	r_2 Read c

The third program differs from the second in that the order of the statements in the last program segment has been reversed. We now have a mixed cycle in $(P \cup A, C - A)$ and the cycle $(p_1 p_2 r_1 r_2 q_2 q_1 p_1)$ See figure 8.

Correct execution is ensured by inserting the delays $p_1 p_2$, $r_1 r_2$, and $q_2 q_1$. We have the paradoxical result that in order to enforce atomic execution of q_1 followed by q_2 we have to issue q_2 first, and delay the issuing of q_1 until q_2 has executed. Still, no locks are required.

3.2. Optimizations

We do not make any assumptions about the mechanism used to secure locks; any synchronization method that yields the effect of locks can be used. An important optimization may be obtained whenever an equivalence class under the relation L consists of just one operation. In that case, it is sufficient to check that the location accessed by that operation is not currently locked by another processor. (In fact, we can even do better: if the operation only reads the location, then it is only necessary to check that no other processor holds a write lock on that location.) This has the same effect as a lock, access, and unlock, so our previous theorems still hold with this optimization. In the even more special case when a location is always accessed by an equivalence class of size 1, no locks on that location are needed at all.

We assumed in the previous section that all locations accessed by operations in the same locking class are locked before any operation in the class executes, and unlocked only after all operations have executed. Instead, we can use a weaker two-phase locking protocol [EGLT]. The locking condition is replaced by the following condition

(2') Let $[u]$ be the equivalence class of operation u under the equivalence relation L . Then no lock is granted on the behalf of an operation in $[u]$ after locks used by operations in $[u]$ have been released.

Thus, the protocol proceeds in two phases, a first one where locks are secured, and a second one where locks are released. A lock to a location is secured before operations in $[u]$ that access the location proceed, and is released only after all such operations have executed (i.e. have been acknowledged). However, memory accesses may start before all locks have been secured, and continue after some locks have been released.

On the other hand, we strengthen the delay condition. The following new condition replaces the previous one:

(1') Assume that uDv . Then if u and v are in the same lock set then v is not issued until u has executed; if u and v belong to distinct lock sets, then the lock for v is not secured until the lock for u is released.

We say that $[v]$ “occurs after” $[u]$ if $[u]=[v]$ and uPv , or if there is a location for which $[u]$ and $[v]$ require conflicting locks, and the lock for $[v]$ is secured after the lock for $[u]$ is released. The “occurs after” relation is a partial order. We say that v “executes after” $[u]$ if $[u] \neq [v]$ and $[v]$ “occurs after” $[u]$, or $[v]=[u]$, and v is executed in memory after u is executed. The relation “executes after” is a lexicographic combination of two partial orders, and hence is a partial order. If uEv then v “executes after” u ; if also $[u] \neq [v]$, then v “executes after” all operations in $[u]$. If uDv then v “executes after” u .

The new definition of execution order has all the relevant properties of the previous one. Thus, all the arguments given in the last section are valid, with the new delay and locking conditions, and the new definition of execution order. In particular, the locking and delay conditions $L(T)$ and T defined in Theorem 4 ensures correct execution, for any two-phase locking policy consistent with L .

3.3. Minimality

We shall prove in this section that the locking and delay conditions defined are optimal. Firstly, any locking relation must contain the tight locking relation T , irrespectively of the delays used. Thus, T is minimal. The delay relation $D(T)$ is not necessarily minimal: It is possible to enforce correct execution with less delays. This, however, requires more operations to be in locking sets; delay pairs are replaced by locking pairs. This is made precise in the following theorem.

Theorem 5: Let L be a locking relation and D be a delay relation such that (L,D) force correctness. Then

- (1) $L \supseteq T$;
- (2) $D \cap T \supseteq D(T) \cap T$; and
- (3) $D \cup L \supseteq D(T) \cup T$.

Proof: (outline) Assume there is a critical pair of a minimal mixed cycle of $(P \cap A, C \cap A)$ that is not contained in $D \cup L$. Then using a similar construction as in Theorem 3, we build an execution order E that is consistent with D and L , such that $(P \cap A, E \cap A)$ has a cycle. Thus, $D \cap \subseteq D \cup L$. Similarly we show that $D \cup \subseteq D \cup L$. It follows that $D(T) \subseteq D \cup L$.

If both $uv \in D \cap \cup D \cup$ and $vu \in D \cap \cup D \cup$ then $uv \in D \cup L$ and $vu \in D \cup L$. Since D is acyclic, this implies that $uv \in L$. It follows that $T \subseteq L$, which proves (1), and $D(T) \cup T \subseteq D \cup L$, which proves (3).

Let uv be a critical pair of a minimal mixed cycle σ in $(P \cap T, C \cap T)$. Since $T \subseteq L$, σ is a minimal mixed cycle of $(P \cap L, C \cap L)$. If uv is not contained in D , then we can build an incorrect execution order that is consistent with D and L . It follows that $D(T) \cap T \subseteq D$, which proves (2). $\square$

4. From Straight Line Code To Programs

4.1. Code Motion

The previous sections addressed the consistency problem from the machine point of view: we considered streams of operations executed sequentially by each processor. The compiler point of view is quite different: The order of operation execution at each processor can be changed, as long as data dependencies are not violated.

The example given in the introduction shows that it is not sufficient to consider only data dependencies within each program segment in order to decide on the validity of code motions. In that example there are no data dependencies within the program segment, yet changing the order of the operations, can cause the set of possible outcomes to change.

The construction of the previous section can be used to decide on the validity of code motion. If there is no delay between operations u and v of the same program segment then the order of execution of these two operations in memory is arbitrary. In particular, we can interchange the order they are issued. Any order of issuing that satisfies the requirement that if uDv then v is not issued until u has been completed is acceptable. The set of possible outcomes will be identical to the set of possible outcomes resulting from the execution of the original program. However, it is important to realize that we enforce consistency with respect to the original program, not the new program. For example, the operations of the following program segments can be executed in arbitrary order:

<i>Segment 1</i>	<i>Segment 2</i>
$p_1 X := 1;$	$q_1 x := X;$
$p_2 Y := 1;$	$q_2 y := Y;$

If initially $X=Y=0$, then any of the four outcomes $x = 0,1, y = 0,1$ is valid. It is valid to reverse the operations in the second program segment, thus obtaining the code given in the first example in the introduction. Again, all four possible outcomes may arise as a result of an execution of the new program. But only three of them are consistent with the new program.

4.2. Partial Orders

In most of our examples, P is a union of chains: operations in each program segment are totally ordered. However, we never actually used any stronger fact than that

- 1) P is a partial order relating every conflicting pair of operations within a program segment; and
- 2) P only relates operations in the same program segment.

We can even relax the first requirement by adding the condition on our delay predicates that either a lock or D must order every pair of conflicting operations within a program segment. We now deal with the second requirement.

4.3. Looping Programs

In the general case our program segments will contain jump statements, due to branch and loop constructs. We represent possible control flow in each program segment by a flow graph (see [ASU]). An execution of a program corresponds to a (possibly self

crossing) path in the flow graph.

One can reduce this general setting to the case of straight-line code by “decoupling” execution of distinct blocks within the same code segment:

One can require that the execution of a new block does not start until memory references issued by instructions in other blocks of this code have been satisfied. For each tuple of blocks, each belonging to another program segment, delays are introduced that enforce correct concurrent execution of these blocks.

It is also possible to do global optimization of delays:

Let uPv if u and v are operations in the same program segment, and there is a path from u to v in the flow graph of this program. P is transitive. However, P may not be acyclic; in particular we may have uPu . To the previously defined conflict relation C we add pairs uu , if u is in a cycle of the flow graph, i.e. if uPu . (This extension allows us to generalize our previous analysis directly, but it may seem strange since u may be a read. Intuitively, the conflict arises because we don’t want one instantiation of a loop to catch up with a previous one.)

If a delay is inserted between operation u and operation v then operation v is not issued as long as there is an executing instance of operation u . A delay relation D *forces sequential consistency* if for any choice of an execution path within each program segment the concurrent execution of these paths is sequentially consistent.

We define an *extended mixed cycle* to be a simple cycle σ in $P \cup C$ that is neither contained in P nor in C . Border pairs are defined as before.

Theorem 6: Let B be the relation define by border pairs of extended cycles. Then B forces sequential consistency.

Proof: Consider the set of execution paths of the program. Let V' be the set of instances of instructions occurring in these paths, and let P' be the partial order defined on V' by these execution paths: $u'P'v'$ if u' occurs before v' on some execution path. Let C' be the relation induced by C on V' . Any simple mixed cycle σ' in $P' \cup C'$ corresponds to an extended mixed cycle σ in $P \cup C$; any border pair $u'v'$ in σ' corresponds to a border pair uv in σ . The claim follows now from theorem 1. $\square$

The definition of B in the last theorem does not seem to be general enough. It might seem to be possible for there to be self-crossing cycles whose border pairs would not be

detected. Note, however, that uCu for any operation u in a cycle of the flow graph. Suppose $(u_0, \dots, u_{n-1}, u_0)$ is a mixed cycle (whose edges are neither contained in P nor in C), u_0u_1 is a border pair of that cycle, and $u_i = u_j$, where $i < j$. Then $(u_0, \dots, u_i, u_{j+1}, \dots, u_{n-1}, u_0)$ is also a mixed cycle with border pair u_0u_1 . We can easily see that any mixed cycle consists of simple cycles and any border pair in that mixed cycle will be a border pair in one of the (simple) extended mixed cycles. Therefore, it is sufficient to consider simple mixed cycles.

The set of cycles can be further reduced by considering only cycles that may correspond to minimal mixed cycles of (P', C') , for some execution paths. In particular one may assume that cycles do not contain more than two consecutive operations that belong to the same block.

5. Some Practical Considerations

5.1. Essential Delays

The border pair relation defined in §2 is transitive. The critical pair relation D defined there may also have triples of operations such that uDv , vDw , and uDw . However, if we enforce the first delay (uDv) and the second delay (vDw), the third delay (uDw) is respected too. Thus it is not necessary to enforce all the delays required by our analysis, but only a subset that spawns the required delay relation.

Let $\bar{D}$ be an acyclic relation defined on a set V . An ordered pair of elements uv is an *essential pair of $\bar{D}$* if the longest path connecting u to v in the graph of $\bar{D}$ has length one. We leave to the reader the proof of the following lemma.

Lemma 7: Let D' be the set of essential pairs for D . Then

- (1) The transitive closure of D' contains D .
- (2) Any relation D'' that is contained in D and fulfills (1) contains D' . $\square$

Once a correct delay relation D has been computed, it is easy to find the essential delays, by running a transitive reduction algorithm [AGU].

A similar result is valid for delay and locking pairs. A delay pair $uv \in D(T)$ is *essential* if either

- (1) $uv \in D(T) - T$, and the longest path from u to v in $D(T) \cup T$ has length one; or
- (2) $uv \in D(T) \cap T$, and the longest path from u to v in $D(T) \cap T$ has length one. Let $D'(T)$ be the set of essential pairs. It is easy to see that $D'(T)$ and T force correctness; $D'(T)$ is the smallest set D of delays such that D and T force correctness.

5.2. Delay Sets

Our use of delays presupposes that a processor issues memory requests without waiting for replies from the previous requests. This is in general true for stores, but not for loads: all instructions following a load are delayed until the load is satisfied. Thus, some delays are automatically enforced by the processor. It is a simple matter to take this fact into consideration in our model: Only those delays in D that are not automatically enforced need to be taken care of.

We assumed that a processor may delay an operation until some arbitrary previous operation has completed. The NYU Ultracomputer and IBM RP3 have a weaker delay mechanism: The issuing of an operation can be delayed until all previously issued operations have terminated. This enforces a delay between the operation and all previous operations. Thus, we can specify a set called the *delay set*. These are operations that are not issued as long as some previous operation executes.

Let I be a set of operations in a program. We associate with I the delay relation δI defined by $u \delta I v$ if $v \in I$ and uPv . (Note that δI may be defined on a superset of I .) Now suppose we define $last(D)$ to be the set $last(D) = \{v : uv \in D\}$. Clearly, $\delta last(D) \supseteq D$. Also, $last(D)$ is the smallest set with this property.

Lemma 8 Let D be a delay relation that enforces sequential consistency. Then so does the delay set $last(D)$. Moreover, $last(D)$ is the smallest delay set that enforces all the delay pairs in D .

The number of delays induced by the delay set δD can be reduced by moving code, consistently with D . Yet, even with code motion, the use of a delay set may result in unnecessary delays. In the following example, b must be delayed until a completes because of the cycle between segments 1 and 2; also, d must be delayed until c completes because of the cycle between segments 1 and 3. But a and b should be able to proceed coindependently with c and d . The weaker delay mechanism would forbid this possibility:

Either b is delayed until c and d complete, or d is delayed until a and b complete.

<i>Segment 1</i>	<i>Segment 2</i>	<i>Segment 3</i>
(a) Read A	(b') Write B	(d'') Write D
(b) Read B	(a') Write A	(c'') Write C
(c) Read C		
(d) Read D		

6. Conclusion

This paper presents a method to enforce efficient and sequentially consistent execution of concurrent processes on a shared-memory multiprocessor when memory access is asynchronous. Our method determines when consecutive operations in the same program segment of a parallel program may execute concurrently, without violating the programmer's view that each segment executes in its given program order. An actual implementation of this method depends on two factors that were not discussed.

First, one should be able to detect data dependencies. This is a problem faced by any optimizing compiler, and sophisticated methods have been developed for that purpose [ASU]. In particular, index analysis to discover data dependencies across loop iterations [AK] are relevant to our purpose. In general, the compiler will not be able to detect existing data dependencies accurately, but will have to assume further dependencies. These extra data dependencies reduce the efficiency of the code produced, but do not affect its correctness.

Second, one has to find all the cycles in a graph. This requires time exponential in the number of nodes in a general graph. However, the graphs arising from program segments have a restricted structure and it seems that the cycles can be detected in polynomial time. A polynomial time algorithm for detection of border pairs of mixed cycles can be given when each program is straight-line; this algorithm extends to the code obtained from high level language program segments with bounded nesting of conditionals.

Our analysis presupposes that the processor has the ability to delay the issuing of an instruction until some previous instruction have been executed. Pipelined processors often have such locking mechanisms. Moreover, the processor detects by itself data dependencies, and uses the locks to enforce correct sequencing of data dependent

operations.

The results of this paper suggest that a more general mechanism is required for processors used in a multiprocessor environment. Rather than checking for data dependencies by itself, the processor should use data dependency information provided by the compiler. For each operation it should be possible to specify one, or a few preceding operations that have to complete before the new operation is executed. The locking mechanism will be based on this information. In a uniprocessor environment, delays would exactly correspond to data dependencies; the extra flexibility could be used to allow pipelining of operations involving memory accesses (pipelined processors usually assume that any two accesses to memory may conflict). In a multiprocessor environment "gratuitous" data dependencies would be added to reflect interprocess interferences.

Acknowledgements

Many thanks to Boris Aronov and to Allan Gottlieb for critical comments on earlier drafts.

References

- [AGU] A. V. Aho, M. R. Garey, and J. D. Ullman, "The transitive reduction of a directed graph", *Siam Journal of Computing*, vol. 1, no. 2, pp. 131-137", 1972.
- [AK] J. R. Allen and K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form", Rice COMP TR84-9, July, 1984.
- [ASU] A. Aho R. Sethi and J. Ullman, *Compilers Principles, Techniques and Tools*, Addison Wesley, Reading, Mass, 1986.
- [BG] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, vol. 13, no. 2, June 1981, pp. 185-221.
- [BSR] P. A. Bernstein, D. W. Shipman, and J. B. Rothnie, "Concurrency control in a system for distributed databases (SDD-1)", *ACM Trans. Database Syst.* 5, 1, March 1980, pp. 18-51.

- [Co] W. Collier, "Principles of Architecture for Systems of Parallel Processes", IBM Tech. Rep. TR00.3100, March 1981.
- [EGK] J. Edler, A. Gottlieb, C.K. Kruskal, K.P. McAuliffe, L. Rudolph, M. Snir, P. Teller, and J. Wilson, "Issues Related to MIMD, Shared-Memory Computers: The NYU Ultracomputer Approach", Proc. 12th Int. Conf. on Computer Architecture, 126-135, Boston, June 1985.
- [EGLT] K. Eswaran, J. Gray, R. Lorie, and I. Traiger, "The notions of consistency and predicate locks in a database system", *CACM*, vol. 19, no. 11, Nov. 1976.
- [GGK] A. Gottlieb, R. Grishman, C.K. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing a MIMD, Shared-memory Parallel Computer", *IEEE Trans. on Computers* C-32 (1983), 175-189.
- [KP] H. T. Kung and C. H. Papadimitriou, "An Optimality Theory of concurrency control for Databases", Proc. ACM-SIGMOD International Conference on Management of Data, 1979, 116-126.
- [La] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", *IEEE Trans. on Computers* C-28 (1979), 690-691.
- [LF] N. A. Lynch and M. J. Fisher, "Semantics of Concurrent Computations", *Theoretical Computer Science* 13, 1 (1981), 17-43.
- [PBG] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton and J. Weiss, "The IBM Research Parallel Processor prototype (RP3): Introduction and Architecture", Proc. IEEE 1985 International Conference on Parallel Processing, 764-771.
- [UW] E. Upfal and A. Wigderson, "How to share memory in a distributed system", Proc. IEEE 1984 Conference on Foundations of Computer Science, pp. 171 - 179.

NYU Ultracomputer

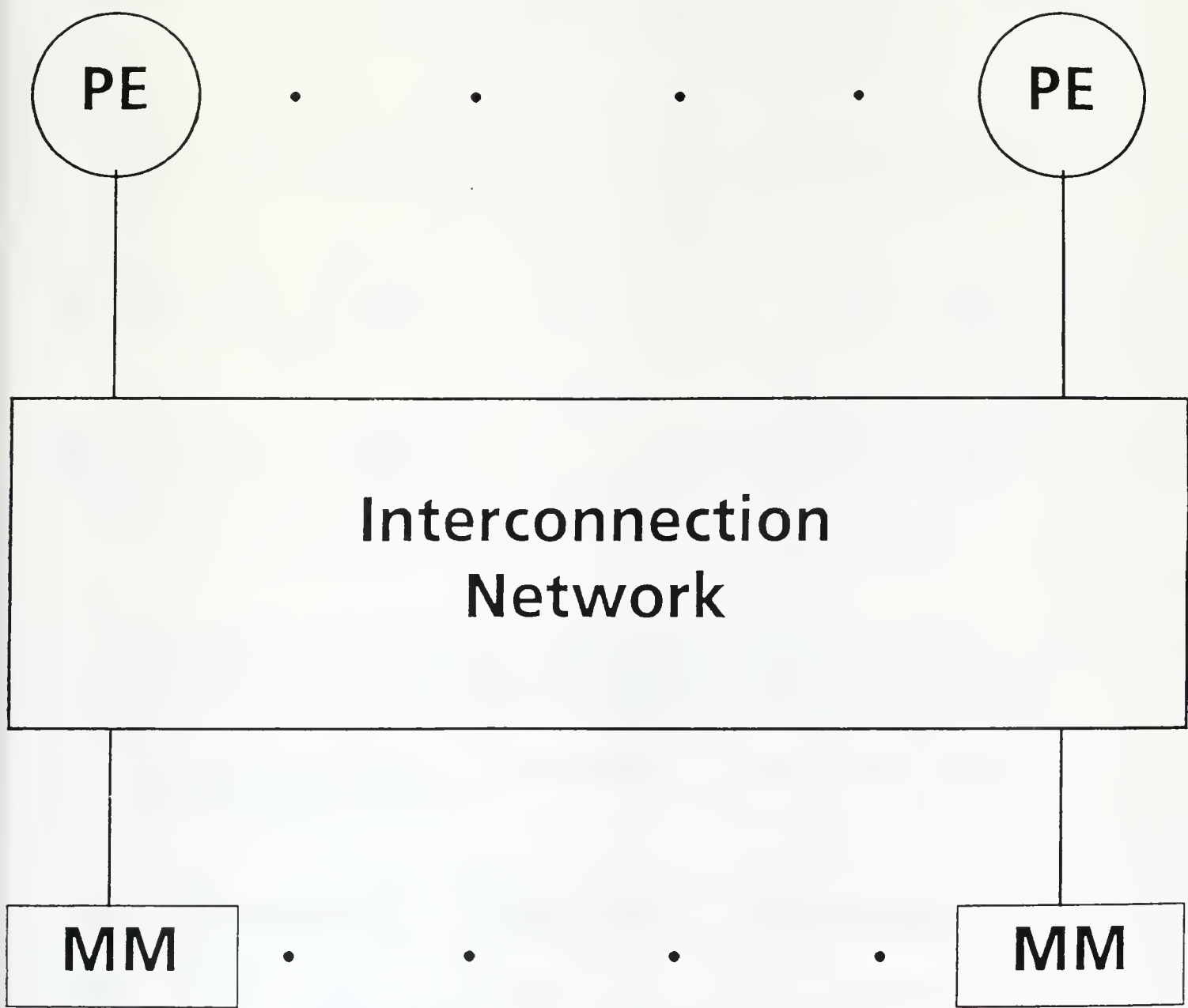


Figure 1

Initially,

$$a = b = 0$$

Segment 1

$$a := 1$$

$$b := 1$$

Segment 2

$$\text{my\_}b := b$$

$$\text{my\_}a := a$$

Want execution to be equivalent to some interleaving of Segment 1 and Segment 2.

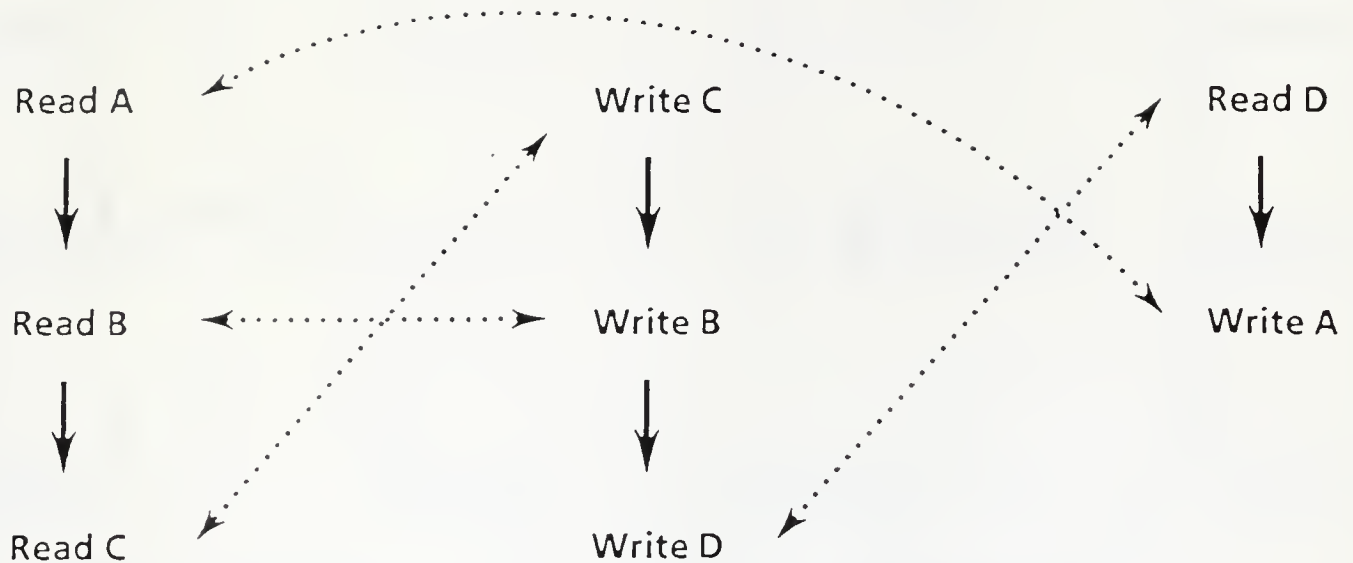
If operations of either segment are executed out of order, we may get $\text{my\_}a = 0$ and $\text{my\_}b = 1$ which is not equivalent to any interleaving.

Figure 2

Segment 1

Segment 2

Segment 3



One of the three mixed
cycles is:
(Read B, Read C, Write C,
Write B, Read B)

Figure 3

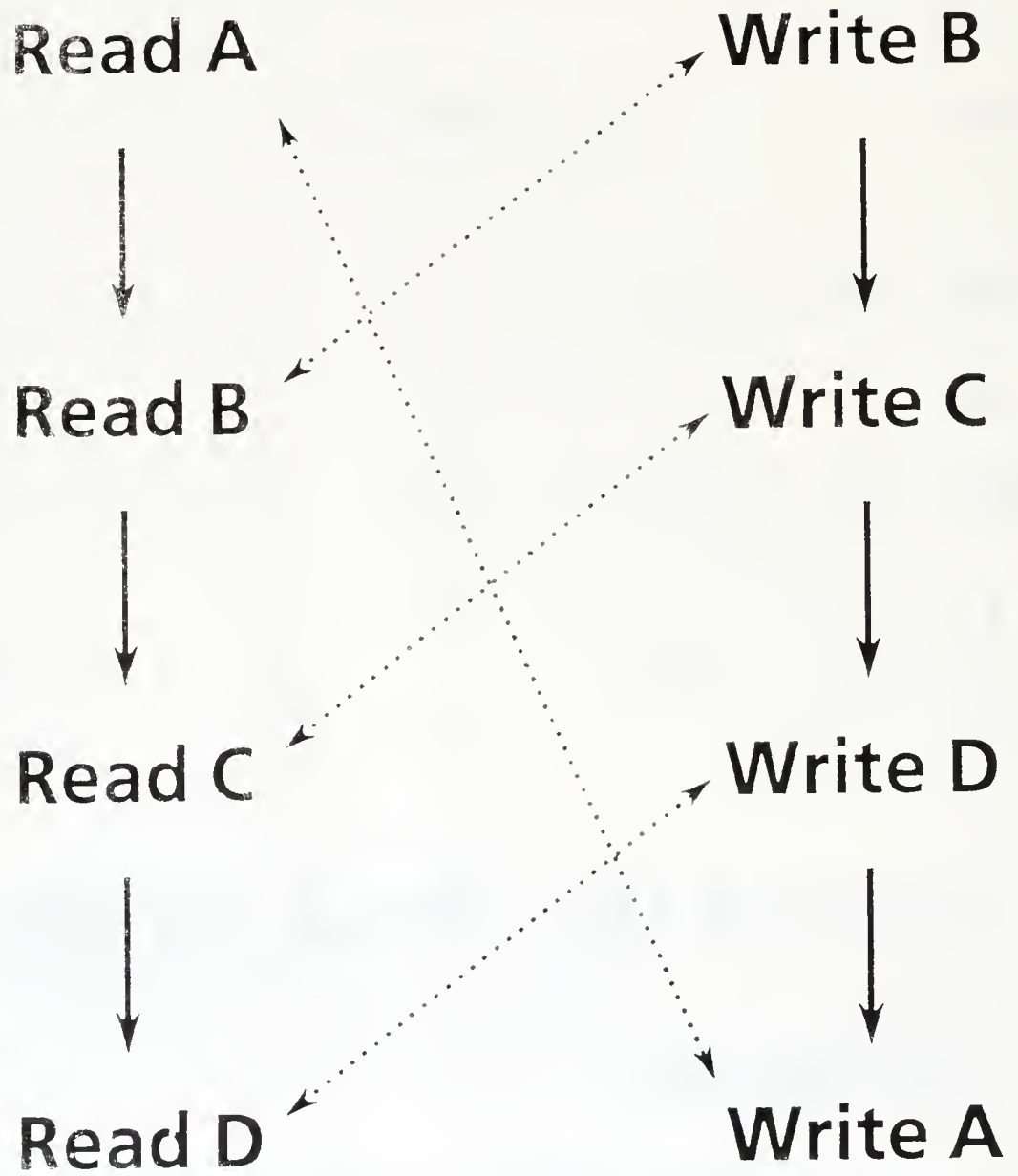


Figure 4

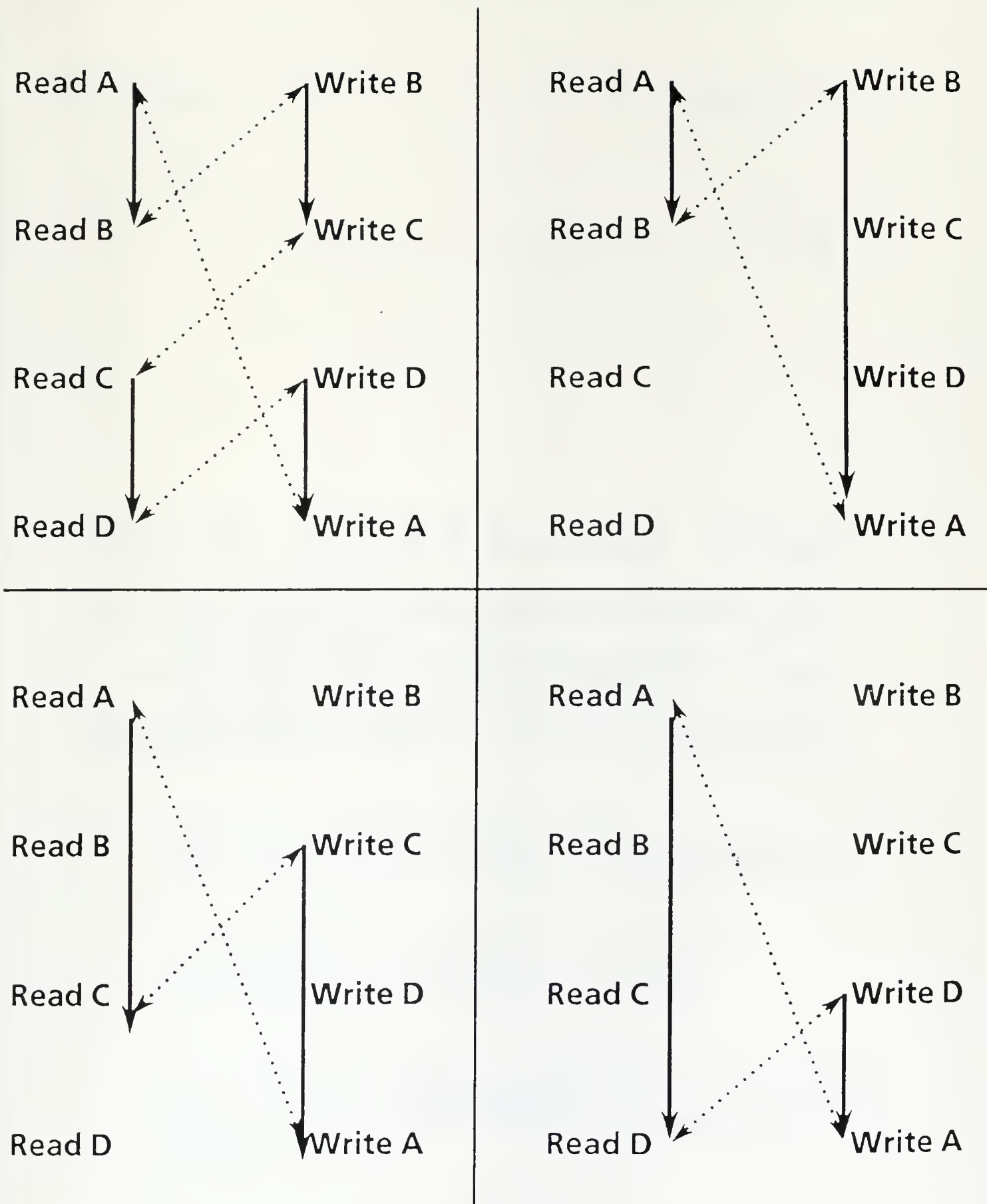
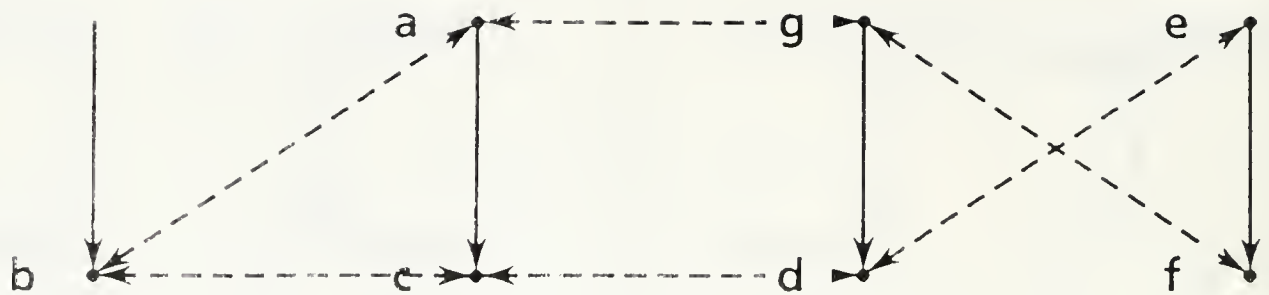


Figure 5

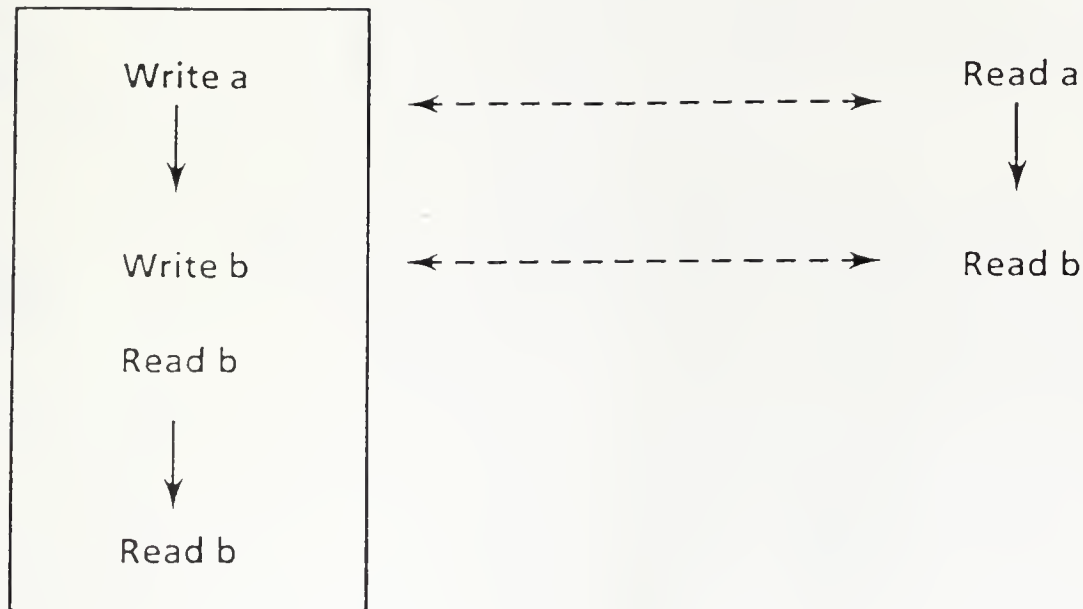


Cycle (a,b,c,d,e,f,g,a) is not minimal because a and c are not consecutive. (a,c,b,a) is a shorter cycle as is (a,c,d,e,f,g,a) .

Figure 6

Segment 1

Segment 2



Boxes indicate atomic blocks.

Mixed cycle in $(P \cap A, C \cap A)$

$(W1(b), R1(b), R1(a), W1(a), W1(b))$

$(P \cap A, C \cap A)$ has a cycle
 $(W1(b), W1(a), R2(a), R2(b), W1(b))$.

So $W1(a)$ and $W1(b)$ are tight
 but $R2(a)$ and $R2(b)$ are not.

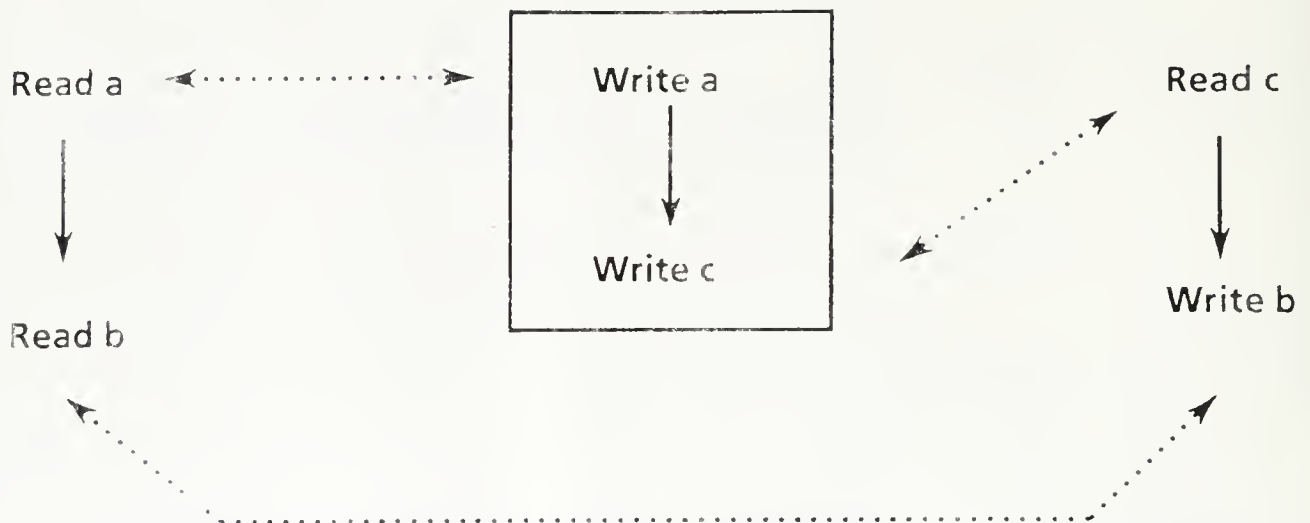
So, lock a and b for $W1(a)$ and $W1(b)$.
 Delay $R1(a)$ for $R1(b)$. Delay $R2(b)$ for $R2(a)$.

Figure 7

Segment 1

Segment 2

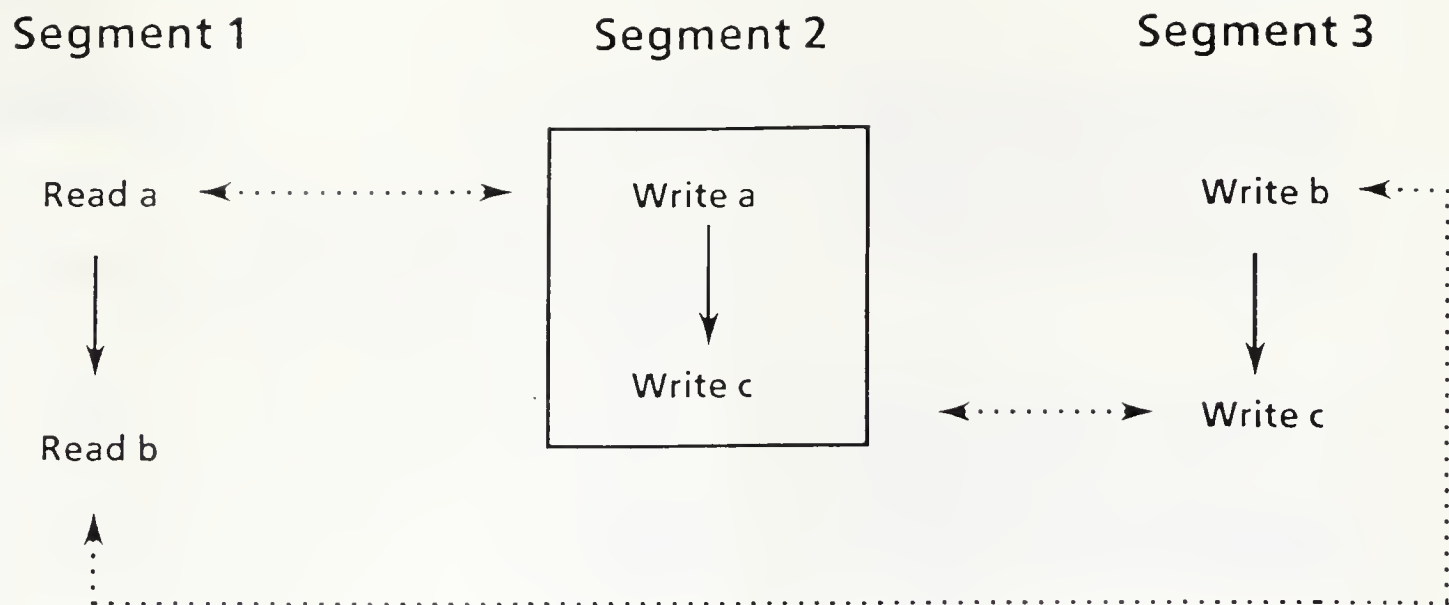
Segment 3



No mixed cycles in (PUA, C-A)
or (PUA, C-A).

No delays or locks needed.

Figure 8



Mixed cycle in (PUA, C-A)

(W3(b), R3(c), W2(c), W2(a), R1(a) R1(b), W3(b)).

Delays shown on Figure 9b.

Note that this reverses their order.

Figure 9a

Segment 1

Read a



Read b

Segment 2

Write a



Write b

Segment 3

Write b



Read c

Delays to handle mixed cycle of **Figure 9a**.

Notice that in segment 2, delay **write a** until **write b** occurs.

Figure 9b

FOURTEEN DAYS

A fine will be charged for each day the book is kept overtime.

GAYLORD 142			PRINTED IN U S A

NYU COMPSCI TR-206 c.2
Shasha, Dennis
Efficient and correct
execution of parallel
programs that share memory

NYU COMPSCI TR-206 c.2
Shasha, Dennis
Efficient and correct
execution of parallel
programs that share memory

LIBRARY
N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012

